OCR Computer Science A Level

2.3.1 Analysis, Design and Comparison of Algorithms

Advanced Notes

Analysis of Algorithms

When developing an algorithm there are two different things to check:

- Time Complexity
- Space Complexity

Time of Complexity

The time complexity of an algorithm is how much time it requires to solve a particular problem. The time complexity is measured using a notation called big-o notation, which shows the effectiveness of the algorithm. It shows an upper limit for the amount of time taken relative to the number of data elements given as an input. This is good because it allows you to predict the amount of time it takes for an algorithm to finish given the number of data elements.

You can think of this as a graph, as the number of data elements entered against the time taken to complete the algorithm. This will be helpful for showing the relationships between time and the number of elements inputted. These are shown below.

Big-O notation is written in the form `O(n)`, where n is the relationship between n: the number of inputted entities, and `O(n)` is the time relationship. Below are examples of different big o notations:

Big O Notation	Name	Description
$O(1)$	Constant time complexity	The amount of time taken to complete an algorithm is independent from the number of elements inputted.
$O(n)$	Linear time complexity	The amount of time taken to complete an algorithm is directly proportional to the number of elements inputted.
$O(n^2)$	Polynomial time complexity (example)	The amount of time taken to complete an algorithm is directly proportional to the square of the elements inputted.
$O(n^n)$	Polynomial time complexity	The amount of time taken to complete an algorithm is directly proportional to the elements inputted to the power of n
$O(2^n)$	Exponential time complexity	The amount of time taken to complete an algorithm will double with every additional item.
$O(\log n)$	Logarithmic time complexity	The time taken to complete an algorithm will increase at a smaller rate as the number of elements inputted.

When calculating the time complexity, you should think logically through the algorithm.
Below are a few examples:

<table>
<tr><th>Algorithm example</th><th>Big-O notation</th></tr>
<tr><td>This examples is unrelated to the number of items inputted:

For example:

<code>print(“hello”)</code></td><td>Constant Time:

This will always take the same amount of time to complete regardless of the number of values inputted.</td></tr>
<tr><td>This example is directly proportional to the number of items inputted:

For example:

<code>inputtedValue = [a,b,c….n]</code>

<pre>for i in range(len(inputedValue)):
     print(“hello””)</pre></td><td>Linear Time Complexity:

The time taken to complete the algorithm is related to the number of items inputted

As you can see, the number of operations completed was proportional to the inputted value.</td></tr>
<tr><td>This example is proportional to the number of items inputted to the power of n:

For example:

<code>inputtedValue = [a,b,c….n]</code>

<pre>For i in range(len(inputedValue)):
     For i in range(len(inputedValue)):
            print(“hello””)</pre></td><td>Polynomial Time Complexity:

The time taken to complete the algorithm is proportional to the number of items inputted to the power of n, below is an example of O(n^2).

As you can see the power given to the polynomial is the same as the number of embedded for loops.</td></tr>
<tr><td>This example is exponentially proportional to the number of items inputted:

For example:

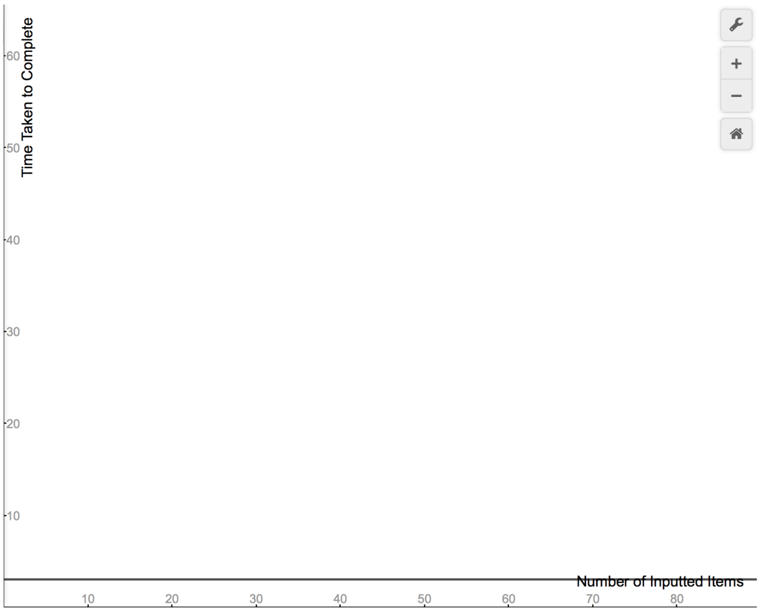
Recursive algorithms that solve a problem of size N by recursively solving two smaller problems of size N-1.</td><td>Exponential Time Complexity:

The time taken to complete the algorithm is proportional to 2 to the power of the number of items inputted.

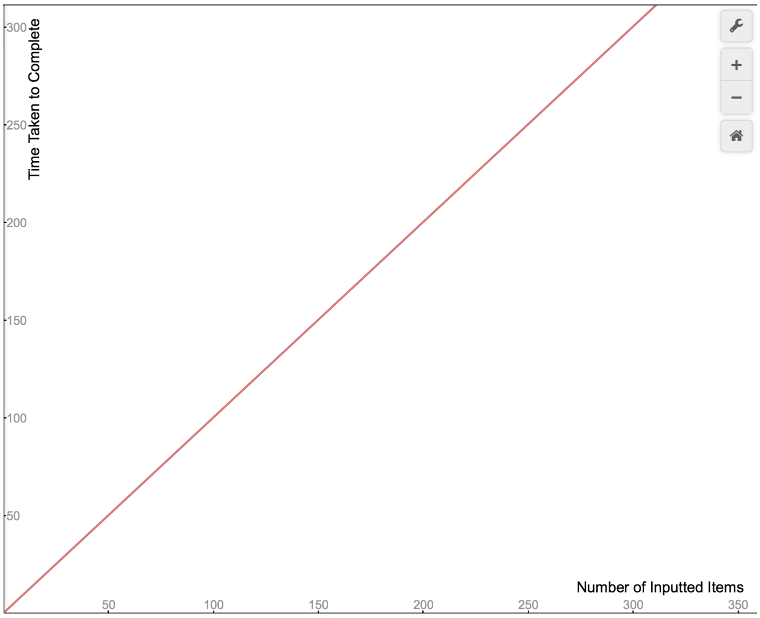
This is common with recursive algorithms solving two smaller problems of size n-1.</td></tr>
</table>

This example is logarithmically related to the number of items inputted (it's important to understand logs for this, they will be explained later on): For example: A divide and conquer algorithm is a good example of this, the number of items you have to search through gets halved every time.	Logarithmic Time Complexity: Logarithms are explained below.

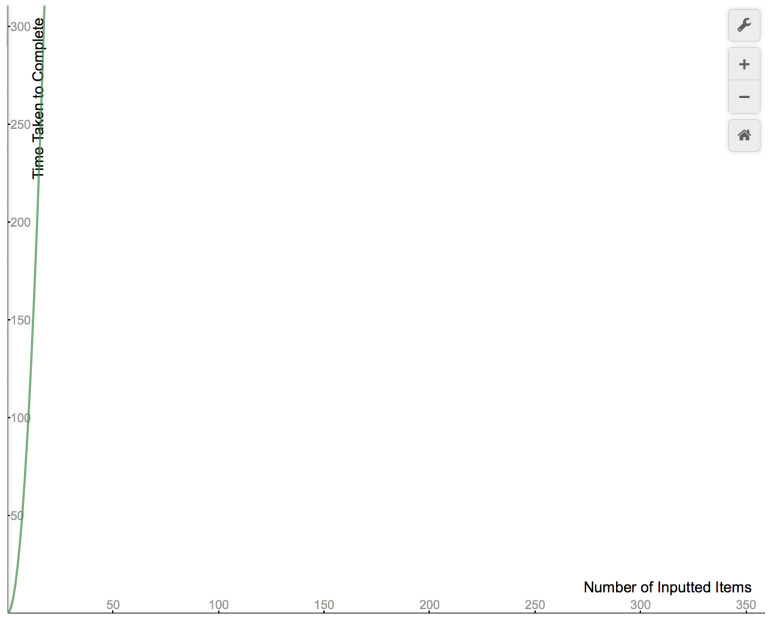
What a Constant Time Complexity graph looks like:



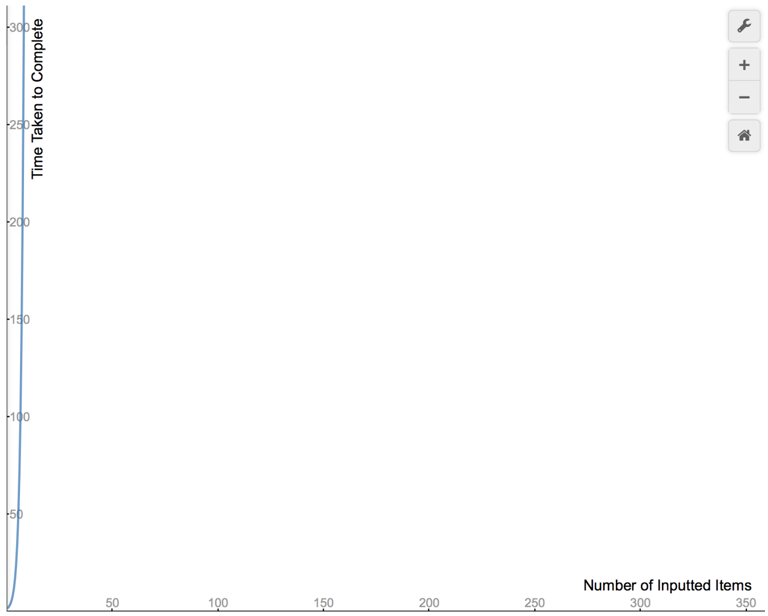
What the Linear Time Complexity graph looks like:



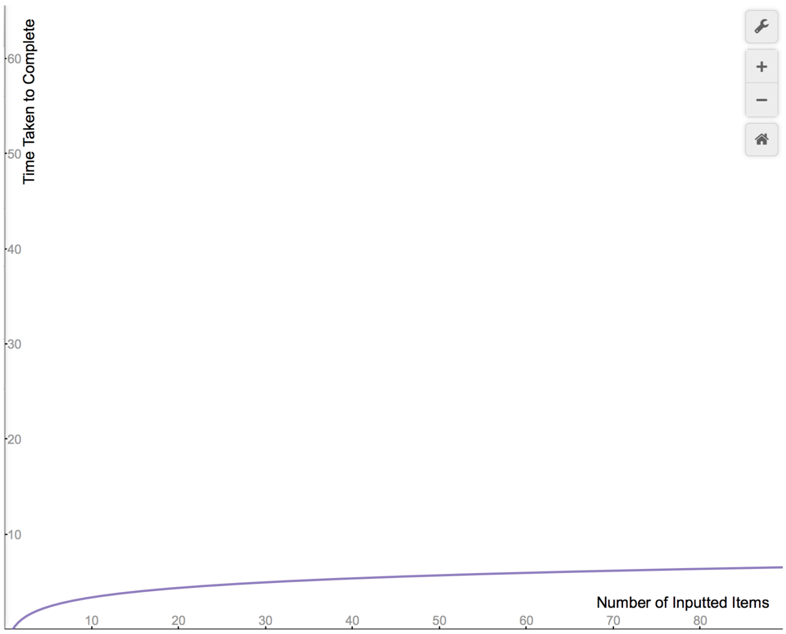
What a Polynomial Time Complexity graph looks like:



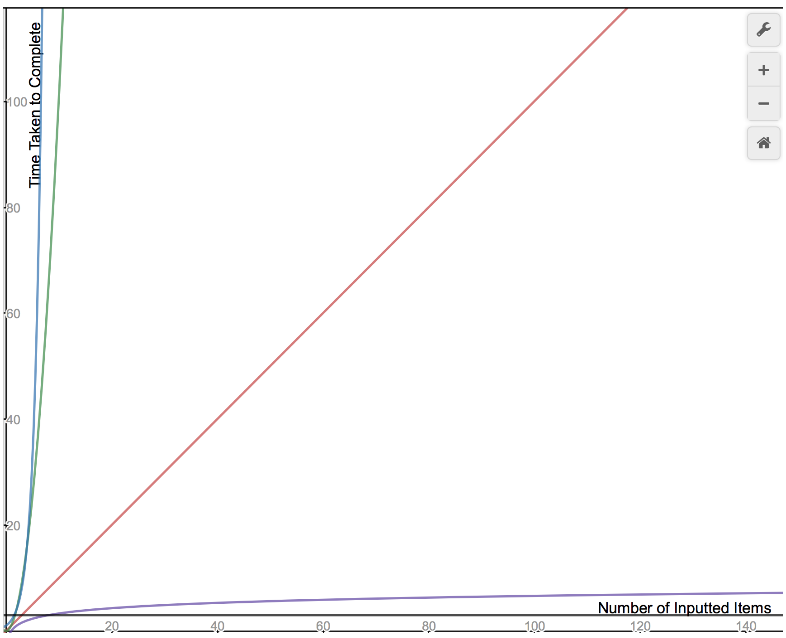
What an Exponential Time Complexity graph looks like:



What a Logarithmic Time Complexity graph looks like:



A comparison of the different complexities:



As you can see from the graph, the best time complexity for an algorithm as the number of inputted items increases is the linear time complexity. The order goes:

Worst

O(2^n) O(n^2)

O(n log(n))

O(n)

O(log(n))

O(1)

Best

Logarithms

A logarithm is the inverse of an exponential, an operation that determines how many times a certain number (base) is multiplied by itself to reach another number. It might help to check the extra resources for more information on this.

An example is shown below:

x	y = log(x)
1 (2^0)	0
8 (2^3)	3
1024 (2^{10})	10

Space Complexity

The space complexity of an algorithm is the amount of storage the algorithm takes. Space complexity is commonly expressed using Big O ($O(n)$) notation. Algorithms store extra data whenever they make a copy, this isn't ideal. When working with lots of data, it's not a good idea to make copies. As this will take lots of storage which is expensive.

Analysing algorithms based on these properties

Time complexity and space complexity are the most important things to keep in mind when you're analysing the effectiveness of a program. The time and space complexity have no priority, you need to decide which one is more important to you at the time you design the algorithm.

Designing Algorithms

An algorithm is a series of steps that completes a task. When you design an algorithm your main objective is to complete a task, the next objectives are to get the best time complexity and the best space complexity. When you try to minimise the time and space complexity you might get conflicted thinking about which one of the two complexities are more important. It is entirely dependant on the situation, below are some examples:

When developing an algorithm for manipulating data in a large database:

- If you have a lot of data but need the data to be processed quickly, say for a future update, then you'd pay more attention to the time complexity rather than the space complexity.
- If you have a lot of processing power then your time complexity isn't as important as you might think, therefore you would focus on the space complexity to make sure you aren't wasting lots of data often.

To reduce the space complexity, you make sure you perform all of the changes on the original pieces of data. To reduce the time complexity, try to reduce the number of embedded loops as possible. Try to reduce the number of items you have to complete the operations on, for example the divide a conquer algorithm accomplishes this and results in logarithmic time complexity.

Comparison of Algorithms

The exam board will mostly compare the time complexity. Occasionally they will mention space complexity although it's important to just understand the smaller the space complexity the better the algorithm is.

Linear Search Algorithm

A linear search algorithm is an algorithm which traverses through every item one at a time until it finds the item its searching for, below is the pseudocode for the linear search algorithm. The Big-O notation for a linear search algorithm is O(n).

```
Function linearSearch(list, item)
     index = -1
     i = 0
     found = False
     while i < length(list) and found = False
          if list[i] = item then
               index = i
               found = True
          endif
          i = i + 1
     endwhile
     return index
endfunction
```

As you can see, the linear search algorithm has a single while loop in it, this is why it's a linear time complexity algorithm

Binary Search Algorithm

A binary search algorithm is a divide and conquer algorithm, this means it splits the list into smaller lists until it finds the item it's searching for, since the size of the list is halved every time it's a Big-O notation of O(log(n)).

```
function binarySearch(list, item)
     found = False
     index = -1
     first = 0
     last = length(list) - 1
     while first <= last and found = False
          midpoint = int ( first + last) / 2 )
          if list[midpoint] = item then
               found = True
               index = midpoint
          else
               if list[midpoint] < item then
                    first = midpoint + 1
               else
                    last = midpoint - 1
               endif
          endif
     endwhile
     return index
endfunction
```

Bubble Sort Algorithm

The bubble sort algorithm passes through the list evaluating pairs of items and ensuring the larger value is above the smaller value. It has a polynomial Big-O notation, $O(n^2)$.

Below is the algorithm:

```
function bubbleSort(list, item)
     found = False
     i = 0
     while found = False          and i < length(item)
          if list[i] > list[i+1]
               temp = list[i]
               list[i] = list[i+1]
               list[i+1] = temp
          endif
          i = i +1
     endwhile
     return list
endfunction
```

Synoptic Link

The algorithms covered here are explained in more detail in the notes for **2.3.1 Sorting Algorithms** and **2.3.1 Searching Algorithms**.